Programozási nyelvek és módszerek

11. ELŐADÁS – PROGRAMKÖNYVTÁRAK

Párhuzamosság folytatás

Várakozás

Egy időre felfüggeszthető a taszk a delay utasítással:

- `delay 2.4;`

A delay után valós típusú érték van, a másodperc megfelelője

Legalább ennyi ideig nem fut a taszk

Szimulációkban gyakran használjuk

Használjuk szinkronizációs utasításokban is

Taszk elindulása és megállása

```
procedure P is
     task T;
     task body T is … begin  … end T;
begin
     -- Itt a T is indul
end P;
-- P bevárja T-t
```

Szülő egység

Az a taszk / alprogram / könyvtári csomag / blokk, amelyben deklaráltuk

Elindulás: a szülő deklarációs részének kiértékelése után, a szülő első utasítása előtt

A szülő nem ér véget, amíg a gyerek véget nem ér

- Függőségi kapcsolatok

Taszk befejeződése

Bonyolult szabályok

Fogalmak: komplett, abortált, terminált

Például:

- elfogytak az utasításai (akár kivétel miatt)
- és a tőle függő taszkok már terminálták

`T'Callable`

`T'Terminated`

Ha több egyforma szál kell

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Háromszálú is

      task Egyik;

      task body Egyik is

      begin

            for i in 1..1000 loop
                Put_Line(„Egyik!");
            end loop;

      end Egyik;
```

Ha több egyforma szál kell

```
      task Másik;

      task body Másik is
      begin
            for i in 1..1000 loop
                  Put_Line(„Másik!");
            end loop;
      end Másik;

begin

      for i in 1..1000 loop
            Put_Line("Viszlát!");
      end loop;

end Háromszálú;
```

Taszk típussal

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Háromszálú is
          task type Üdvözlő;
          task body Üdvözlő is
          begin
                loop  Put_Line("Szia!");    end
loop;
          end Üdvözlő;
          Egyik, Másik: Üdvözlő;
begin
          loop  Put_Line("Viszlát!");   end loop;
end Háromszálú;
```

Taszk típus

Taszkok létrehozásához

- Ugyanazt csinálják, ugyanolyan utasítássorozatot hajtanak végre
- Saját példánnyal rendelkeznek a lokális változókból

Korlátozott (limited) típusok

Cím szerint átadandó, ha alprogram-paraméter

Programegységek

- Mint a taszk programegységek
- Specifikáció és törzs különválik
- Nem lehet könyvtári egység
  (be kell ágyazni más programegységbe)

```
procedure Haromszal is
  task type Szalak(I : Integer); -- a szál azonosítója
  task body Szalak is
  begin
       for j in 1..1000 loop
              put(I); new_line;
       end loop;
  end Szalak;
  X:Szalak(1);
  Y:Szalak(2);
begin
   for i in 1..100 loop
         put_line("foprogram");
   end loop;
end Haromszal;
```

Szinkron kommunikáció finomítása

Hívott (szolgáltató) részéről

- Meddig várjon?
- Több igényt is kiszolgálhat
- Feltétel lehetősége jó lenne
- Mindörökké fusson?

Hívó részéről

- Meddig várjon?

Igény a programnyelvi támogatásra!

A select utasítás

A hívóban is és a hívottban is szerepelhet – lehetséges magatartásformák támogatása

A randevúk szervezéséhez segítség

- Többágú hívásfogadás
- Feltételhez kötött hívásfogadás
- Időhöz kötött hívásfogadás
- Feltételes hívásfogadás
- Termináltatás

hívott

- Időhöz kötött hívás
- Feltételes hívás

hívó

Többágú hívásfogadás – hívottban

```
select
       accept E1 do ... end E1;
or
       accept E2;
or
       accept E3( P: Integer ) do ... end E3;
end select;
```

A hívott választ egy hívót valamelyik várakozási sorból

Ha mindegyik üres, vár az első hívóra: bármelyik randevúra hajlandó

Több műveletes monitor

```
task body Monitor is
   Adat: Típus; ...
begin
   loop
      select
            accept Művelet_1( ... ) do ... end;
      or
            accept Művelet_2( ... ) do ... end;
      ...
      end select;
  end loop;
end;
```

Végtelen taszk

Az előző taszk végtelen ciklusban szolgálja ki az igényeket

Sosem ér véget

A szülője sem ér véget soha

Az egész program „végtelen sokáig” fut

Ha már senki sem akarja használni a monitort, akár le is állhatna...

Termináltatás

A select egyik ága lehet terminate utasítás

Ez azt jelenti, amit az előbb mondtunk

- ha már senki nem akarja használni, akkor terminál jon

Akkor „választódik ki” a terminate ág, ha már soha többet nem jöhet hívás az alternatívákra

A terminate használata

```
select
     accept E1;
or
     accept E2( ... ) do ... end;
or
     accept E3 do ... end;
or
     terminate;
end select;
```

```
task body Monitor is
   Adat: Típus; ...
begin
   loop
        select
                accept Művelet_1( ... ) do ... end;
        or
                accept Művelet_2( ... ) do ... end;
        ...
        or
                terminate;
        end select;
  end loop;
end;
```

Feltételhez kötött hívásfogadás

A select egyes ágaihoz feltétel is rendelhető

Az az ág csak akkor választódhat ki, ha a feltétel igaz

```
select
     accept E1 do ... end;
or
     when Feltétel => accept E2 do ... end;
...
end select;
```

A feltétel használata

A select utasítás végrehajtása az ágak feltételeinek kiértékelésével kezdődik

- Zárt egy ág, ha a feltétele hamisnak bizonyul ebben a lépésben
- A többi ágat nyitottnak nevezzük
  - A nyitott ágak közül nem-determinisztikusan választunk egy olyat, amihez van várakozó hívó
  - Ha ilyen nincs, akkor várunk arra, hogy valamelyik nyitott ágra beérkezzen egy hívás
  - Ha már nem fog hívás beérkezni, és van terminate ág, akkor lehet terminálni (terminálási szabályok)

A feltételek csak egyszer értékelődnek ki

Időhöz kötött hívásfogadás

Ha nem vár rám hívó egyik nyitott ágon sem, és nem is érkezik hívó egy megadott időkorláton belül, akkor hagyjuk az egészet a csudába...

Tipikus alkalmazási területek

- timeout-ok beépítése (pl. holtpont elkerülésére)
- ha rendszeres időközönként kell csinálni valamit, de két ilyen között figyelni kell a többi taszkra is

```
select
     accept E1;
or
     when Feltétel => accept E2 do ... end;
or
     delay 3.0;
end select;
```

```
loop
      select
            when Feltétel => accept E do ... end;
      or
            delay 3.0;
      end select;
end loop;
```

Rendszeresen újra ellenőrzi a feltételt, hátha megváltozott (egy másik folyamat hatására)

Feltételes hívásfogadás

Ha most azonnal nem akar velem egy hívó sem randevúzni, akkor nem randevúzom.

- Nem várok hívóra, futok tovább...

Tipikus alkalmazási területek

- valamilyen számítás közben egy jelzés megérkezését ellenőrzöm
- rendszeres időközönként randevúzni vagyok hajlandó, de egyébként valami mást csinálok
- holtpont elkerülése (kiéheztetés veszélye mellett)

```
select
     accept E1;
or
     when Feltétel => accept E2 do ... end;
else
     Csináljunk_Valami_Mást;
end select;
```

Ha minden nyitott ág várakozási sora üres, akkor csináljunk valami mást

```
select
      accept E1;
or
      when Feltétel => accept E2 do ... end;
else
      delay 3.0;
end select;
```

Nem ugyanaz, mint az időhöz kötött hívásfogadás!

```
loop
        select
                accept E1;
        else
                null;
                -- busy-waiting
        end select;
end loop;
```

Amíg a szükséges helyzet elő nem áll, a folyamat dinamikusan várakozik, azaz folyamatosan vizsgálja, hogy a kívánt feltételek teljesülnek-e.

Hátránya, hogy a folyamat várakozás közben foglalja a processzoridőt.

Időhöz kötött hívás

A hívóban szerepel „or delay” a select utasításban

Ilyen select-ben csak két ág van

- a hívás
- és az időkorlát – eddig vár a randevúra

```
select
      Lány.Randi;
or
      delay 3.0;
end select;
```

Feltételes hívás

A hívóban szerepel else ág a select utasításban

Ilyen select-ben csak két ág van

- a hívás
- és az alternatív cselekvés

```
select
    Lány.Randi;
else
    Csináljunk_Valami_Mást;
end select;
```

Termelő-fogyasztó feladat

Egy folyamat adatokat állít elő, egy másik adatokat dolgoz fel

Kommunikáció: egy adatokat tartalmazó sor

- Termelő => sor => fogyasztó
- Levelesláda, futószalag; a Tároló általánosítása
- Aszinkron kommunikáció

Korlátos vagy korlátlan sor (buffer)

Általánosítás: több termelő, több fogyasztó

Termelő folyamat

```
task type Termelő ( Sor: Sor_Access );
task body Termelő is
     Adat : Típus;
begin
     …
     loop
          Előállít( Adat );
          Betesz( Sor.all, Adat );
     end loop;
end Termelő;
```

Fogyasztó folyamat

```
task type Fogyasztó ( Sor: Sor_Access );
task body Fogyasztó is
     Adat: Típus;
begin
     …
     loop
          Kivesz( Sor.all, Adat );
          Feldolgoz( Adat );
     end loop;
end Fogyasztó;
```

A sor (buffer, FIFO)

Több folyamat használja egyidőben

Közös változó

- Védeni kell, például monitorral (blokkolhatjuk a termelőket/fogyasztókat)
  - Ha üres, nem lehet kivenni belőle
  - Ha tele van (korlátos eset), nem lehet beletenni
  - Egy időben egy műveletet lehet végrehajtani
    - Ha több elem van benne, esetleg lehet
      Betesz || Kivesz

Szálbiztos sor

Osztott_Sorok sabloncsomag:

- Sor típus
- Szálbiztos (taszkok közötti kommunikációhoz)
- Támaszkodik a Sorok-ra
- Plusz szinkronizációs aspektus (monitor)

Sorok sabloncsomag:

- Sor típus
- Hagyományos, nem szálbiztos

Taszkok elrejtése (1)

```
generic
       type Elem is private;
package Osztott_Sorok is
       type Sor(Max_Méret: Positive) is limited private;
       procedure Betesz( S: in out Sor; E: in Elem );
       procedure Kivesz( S: in out Sor; E: out Elem );
       ….–üres, tele
private
       task type Sor( Max_Méret: Positive) is
              entry Betesz( E: in Elem );
              entry Kivesz( E: out Elem );
       end Sor;
end Osztott_Sorok;
```

megvalósítás: monitorral

Taszkok elrejtése (2)

```
with Sorok;
package body Osztott_Sorok is
    procedure Betesz( S: in out Sor; E: in Elem ) is
    begin
       S.Betesz(E);
    end;
    procedure Kivesz( S: in out Sor; E: out Elem ) is
    begin
       S.Kivesz(E);
    end;
    task body Sor is … end Sor;
end Osztott_Sorok;
```

Taszkok elrejtése (3)

```
task body Sor is
     package Elem_Sorok is new Sorok(Elem);
     use Elem_Sorok;
     S: Elem_Sorok.Sor(Max_Méret);
begin
     loop
          …
     end loop;
end Sor;
```

Taszkok elrejtése (4)

```
loop
  select
       when Üres(S) => accept Betesz( E: in Elem ) do
                                                Betesz(S,E);
                                           end Betesz;
  or
       when Tele(S) => accept Kivesz( E: out Elem ) do
                                                Kivesz(S,E);
                                          end Kivesz;
  or
       terminate;
  end select;
end loop;
```

Védett típusú korlátos pufferrel

```
protected type Korlátos_Buffer ( Max: Positive ) is
   entry Betesz( X: in Elem );
   entry Kivesz( X: out Elem );
private
   Adatok: Vektor(1..Max);
   Be, Ki: Positive := 1;
end Korlátos_Buffer;
```

```
protected body Korlátos_Buffer is
    entry Betesz( X: in Elem ) when Be-Ki < Max is
    begin
       Adatok(Be) := X;  Be := Be + 1;
    end Betesz;

    entry Kivesz( X: out Elem ) when Be-Ki > 0 is
    begin
       X := Adatok(Ki);  Ki := Ki + 1;
       if Ki > Max then
           Ki := Ki - Max; Be := Be - Max;
       end if;
    end Kivesz;
end Korlátos_Buffer;
```

Az evő filozófusok problémája

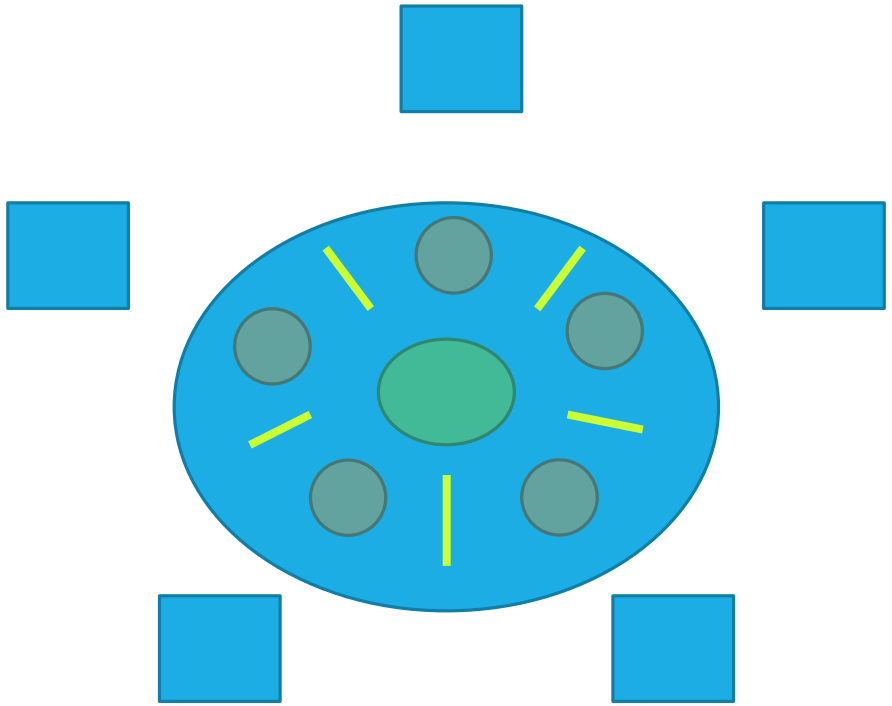
Dijkstra: dining philosophers

Erőforráshasználat modellezése

- Erőforrások: villák
- Folyamatok: filozófusok

Holtpont kialakulása

Éheztetés veszélye



Holtpont

Deadlock

- Ha a folyamatok egymásra várnak – ha egyszerre éheznek meg a filozófusok …

Elkerülés

- Például a szimmetria megtörésével:
  - Erőforrások lekötése rendezés szerint
  - Véletlenszerű időkorlátok
- Ajtó bevezetésével – erre is lehetne példa

Az erőforrások modellezése

```
task type Fork is
        entry Grab;
        entry Put_Down;
end Fork;
task body Fork is
begin
        loop
                accept Grab;
                accept Put_Down;
        end loop;
end Fork;
```

Az erőforrások modellezése(2)

Protected típussal:

```
protected type Fork is
      entry Grab;
          procedure Put_Down;
      private
         Seized : Boolean := False;
 end Fork;
 protected body Fork is
      entry Grab when not Seized is
         begin
            Seized := True;
         end Grab;
      procedure Put_Down is
         begin
            Seized := False;
         end Put_Down;
 end Fork;
```

A filozófusok

```
type Philosopher is
                (Aristotle,
                 Kant,
                 Spinoza,
                 Hegel,
                 Russel);
```

Kell egy véletlenszám-generátor is

A filozófusok

```
task type Person (Id : Philosopher; First, Second :  access Fork );
 task body Person is
        Dice : Generator;   -- random generator
     begin
        Reset (Dice);
        for Life_Cycle in 1..Life_Span loop
           Put_Line (Philosopher'Image (Id) & " is thinking");
           delay Duration (Random (Dice) * 0.100);
           Put_Line (Philosopher'Image (Id) & " is hungry");
           First.Grab;
           Second.Grab;
           Put_Line (Philosopher'Image (Id) & " is eating");
           delay Duration (Random (Dice) * 0.100);
           Second.Put_Down;
           First.Put_Down;
        end loop;
        Put_Line (Philosopher'Image (Id) & " is leaving");
     end Person;
```

Étkező filozófusok

```
Forks : array (1 .. 5) of aliased Fork;
    -- az éhes filozófusok villái
    -- Filozófusok
      Ph_1 : Person (Aristotle, Forks (1)'access, Forks (2)'access);
      Ph_2 : Person (Kant, Forks (2)'access, Forks (3)'access);
      Ph_3 : Person (Spinoza, Forks (3)'access, Forks (4)'access);
      Ph_4 : Person (Hegel, Forks (4)' access, Forks (5)'access);
      Ph_5 : Person (Russel, Forks (1)' access, Forks (5)'access);
   begin
      null; -- a főprogramban csak ülünk és figyelünk ☺
   end;
```

Test_dining_philosophers.adb

- Eredmény a Filozofus_uj.rtf-ben

Véletlenszámok

Álvéletlenszámok (pseudo-random numbers)

Valamilyen (determinisztikus) algoritmussal

Float vagy diszkrét típusú

- Ada.Numerics.Float_Random (csomag)
- Ada.Numerics.Discrete_Random (sablon csomag)

Generator, Reset, Random stb.

Szimulációkhoz

Taszkok esetén: szinkronizáltan

Író-olvasó feladat

Van egy több folyamat által használt erőforrás

Lehet változtatni („írni”)
és lekérdezni („olvasni”)

Az olvasások mehetnek egyidőben

Az írás kizárólagos

A monitornál megengedőbb

Megvalósítás Adában

Taszkok segítségével

- Bonyolult
- Kell pár select bele
- Igazságosság, pártatlanság (fairness)

Védett egységgel

- egyszerű

Író-olvasó probléma

- többen olvashatnak
- egyszerre csak egy írhat

```
protected Változó is
   function Lekérdez return Adat;
   procedure Beállít (Új: in Adat);
private
   V: Adat;
end Változó;
```

Író-olvasó probléma

- többen olvashatnak
- egyszerre csak egy írhat

```
protected body Változó is
   function Lekérdez return Adat is
      begin return V; end Lekérdez;

   procedure Beállít (Új: in Adat ) is
      begin V := Új; end Beállít;
end Változó;
```

Kiéheztetés

Livelock, starvation

- Nincs holtpont (deadlock)

Fut a rendszer

- De: egy folyamat mindig rosszul jár
- Nem képes elvégezni a feladatát
- Példa: ügyetlen „író-olvasó” megvalósításnál
  - Az olvasók könnyen kiéheztethetik az írókat
- Példa: ügyetlen „evő filozófusok” megvalósításnál
  - Szimmetria megtörése a filozófusok beszámozásával
  - A kisebb sorszámú előnyt élvez a nagyobb sorszámúval szemben

Algol-68

Az Algol-68-ban a vesszővel elválasztott és begin - end közé írt utasítások párhuzamosan futtathatók.

A begin-end addig nem fejeződik be, amíg minden vessző közötti egység - párhuzamos klóz- véget nem ér.

A szinkronizálásra Dijkstra szemaforokat vezettek be (sema).

A szemaforok párhuzamos klózai elé ki kell tenni a par kulcsszót.

```
begin
   sema mutex := level 1;
   bool not finished := true;
      # közösen használt puffer deklarációja #
   proc producer = void:
      while not finished
      do
      down mutex
      # pufferbe egy elemet #
      up mutex
      od;
   proc consumer = void:
      while not finished
      do
      down mutex
      # pufferből egy elemet #
      up mutex
      od;
   par (producer, consumer) ;
end;
```

Delphi

A TThread osztályból képzett alosztályok párhuzamosan végrehajthatók.

A szinkronizálásra és a kommunikálásra a főosztályban létre kell hozni egy eljárást, amely a vezérlésért felelős, majd a synchronize(eljárás_név) segítségével a párhuzamos alosztályok kommunikálását ellenőrizhetjük.

Delphi

A TThread osztály legfontosabb (virtuális) absztrakt művelete az Execute() metódus, melyet minden leszármazottban felül kell definiálni. Ez tartalmazza a szál fő kódját.

Szálat a Create konstruktor meghívásával hozhatunk létre, melynek egy Boolean paramétere van (CreateSuspended), mellyel azt adjuk meg, hogy a szál felfüggesztődjön (true), vagy a létrehozás után azonnal elinduljon (false).

Delphi

```
type TMyThread = class(TThread) protected
          procedure Execute; override;
end;
```

meg kell adni az Execute törzsét …

használata:

```
thread:=TMyThread.Create(false);
// el is indítjuk egyben
```

Delphi

Szemafor:

```
var MySemaphore: TSemaphore;
// szemafor változó deklarálása
begin
MySemaphore := TSemaphore.Create(nil, 5, 5, '');
    // 5 szál számára engedélyezzük az egyidejű
használatot ...
MySemaphore.Acquire;
    //várakozás egy szemaforra (lefoglalás)
    ... // itt dolgozhatunk az osztott erőforráson
MySemaphore.Release; // szemafor elengedése
...
MySemaphore.Free; // szemafor megsemmisítése
end;
```

Delphi

A szinkronizációs esemény egy olyan objektum, amelyet egy szál kiválthat, így jelezve egy másik szálnak, hogy folytathatja a tevékenységét.

```
type
  TMyThread1 = class(TThread)
     protected procedure Execute; override;
  end;
  TMyThread2 = class(TThread)
     protected procedure Execute; override;
  end;
```

Delphi

```
 var MyEvent: TEvent;  // ez lesz az esemény
        Thread1: TMyThread1; Thread2: TMyThread2;
procedure TMyThread1.Execute;
    begin
    ...
      MyEvent.SetEvent; //kiváltjuk az eseményt
     ...
  end;
procedure TMyThread2.Execute;
    begin
    ...
       MyEvent.WaitFor(INFINITE); //várunk az eseményre
    ...
end;
```

Java

A Java nyelvben a folyamatok (szálak) leírására a Thread osztályt használják, az osztály egy objektuma reprezentál egy folyamatot.

- A folyamat törzse a run metódusban leírt kód.

Egy programban tetszőleges számú szálat indíthatunk. Ezeknek megadhatjuk a prioritását, felfüggeszthetjük, újraindíthatjuk, megállíthatjuk.

A szálakat szinkronizálhatjuk a join segítségével.
A kommunikáció osztott változókon keresztül történik, a kölcsönös kizárás biztosítására a synchronized kulcsszó szolgál. Ha egy blokk vagy metódus synchronized akkor a hozzá csatolt monitor biztosítja a kölcsönös kizárást.

Runnable interfész megvalósítása kell, ha nem tud örökölni a Threadtől!

Java

```
class MyThread extends Thread {
 public void run() {
  System.out.println("Helló, ez itt a" + getName()
+ " Thread" );
    }
 }
 public class ExtendedThread {
  static public void main(String args[]) {
      MyThread a, b;
            a = new MyThread();
            b = new MyThread();
      a.start();
      b.start(); }
}
```

Java

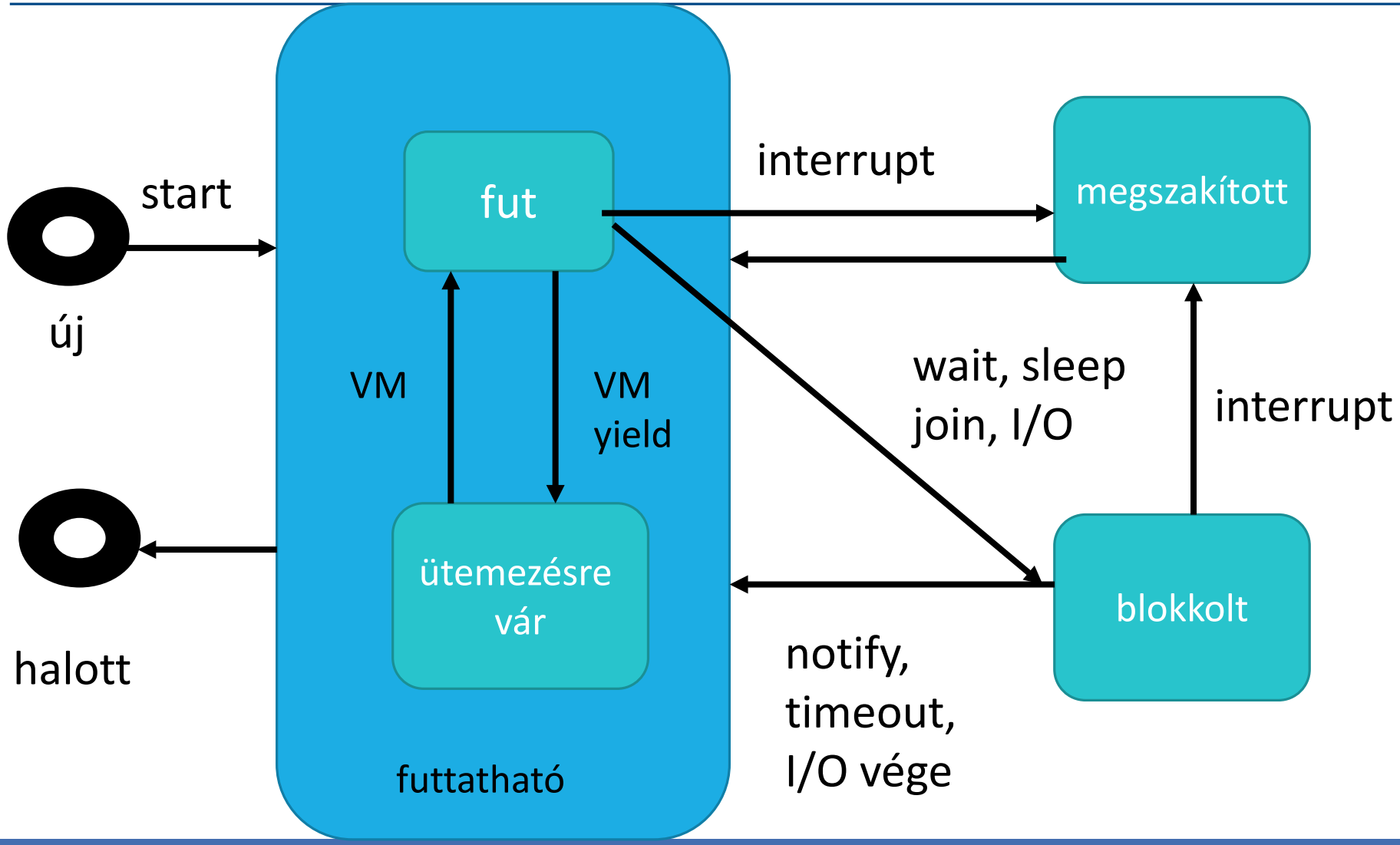
```
class MyThread implements Runnable {
  public void run() {
          System.out.println("Helló, ez itt a " +
              Thread.currentThread().getName() + "        Thread" );
  }
 }

public class RunnableThread {
 static public void main(String s[]) {
   MyThread work2do;
   Thread a, b;
   work2do = new MyThread();
   a = new Thread(work2do);
   b = new Thread(work2do);
   a.start();
   b.start();
   }
 }
```

Java

Szálak állapotai:

- Futó: a VM éppen ezt a szálat futtatja
- Kész (ready): futtatható, éppen áll, hogy egy másik szál futhasson
- Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
- Megszakított: a folyamatoknak jeleket küldhetünk, melyekre megállnak
- Halott: befejeződött a folyamat

Szálak állapotai



Java

Szálcsoportok is létrehozhatók – a ThreadGroup segítségével – csoportszinten is kezelhetők a szálak

Futásvezérlés

- Felfüggesztés – újraindítás
- suspend() - resume() pár, ezt ma már nem ajánlják – holtpontveszély
  - pl. egy szál zárol egy erőforrást, és felfüggesztik …
- Javaslat
  - segédváltozó és a wait() – notify() pár!

Java

- Megszakítás – interrupt()
- Appleteknél a stop() metódusban kell a felfüggesztést is elengedni, ill. a megszakítást meghívni.
- Leállítás
  - volt: stop() – nem biztonságos
  - most: volatile Thread referencia, ami a futó szálra mutat, ha a szálat leállítják, ez null lesz, amit a run() vizsgál
- Szálak összekapcsolása – join() – szinkronizálásra – megvárja, amíg a másik befejeződik (vagy egy adott ideig vár)
- Prioritások kezelése – 10 szint, állítható

Java

Versengés

- Több szál szimultán szeretne ugyanahhoz az erőforráshoz hozzájutni
  - synchronized blokk
    - objektum- és osztályszinten
  - Problémák
    - író/olvasó esetben egyszerre csak egy írhat és olvashat, bár elvben több is olvashatna

Java – A synchronized kulcsszó

Metódusok elé írhatjuk (de pl. interfészekben nem!)

Kölcsönös kizárás arra a metódusra

Kulcs (lock) + várakozási sor

- A kulcs azé az objektumé, amelyiké a metódus
- Ugyanaz a kulcs az összes szinkronizált metódusához
  1. Mielőtt egy szál beléphetne egy szinkronizált metódusba, meg kell szereznie a kulcsot
  2. Vár rá a várakozási sorban
  3. Kilépéskor visszaadja a kulcsot

Java – Szinkronizált blokkok

A synchronized kulcsszó védhet blokk utasítást is

Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon

- `synchronized(obj){...}`

Sokszor úgy használjuk, hogy a monitor szemléletet megtörjük

- Nem az adat műveleteire biztosítjuk a kölcsönös kizárást, hanem az adathoz hozzáférni igyekvő kódba tesszük…
- Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni

Java – wait-notify

Szignálokra hasonlít

Egy feltétel teljesüléséig blokkolhatja magát egy szál

A feltétel (potenciális) bekövetkezését jelezheti egy másik szál

Alapfeladat

- termelő - fogyasztó
  (korlátos) bufferen keresztül kommunikálnak
  - egy termelő, egy fogyasztó
  - több termelő, több fogyasztó

Java – wait-notify

Minden objektumhoz tartozik a sima kulcshoz tartozó várakozási soron kívül egy másik, az ún. wait-várakozási sor

- A wait() hatására a szál bekerül ebbe
- A notify() hatására az egyik várakozó kikerül belőle
- A wait() és notify() hívások csak olyan kódrészben szerepelhetnek, amelyek ugyanazon az objektumon szinkronizáltak
- `synchronized(obj){ ... obj.wait(); ... }`

Java – wait-notify

A szál megszerzi az objektum kulcsát, ehhez, ha kell, sorban áll egy ideig (synchronized)

- A wait() hatására elengedi a kulcsot, és bekerül a wait-várakozási sorba
- Egy másik szál megkaparinthatja a kulcsot (kezdődik a synchronized)
- A notify() vagy notifyAll() metódussal felébreszthet egy vagy az összes wait-es alvót, aki bekerül a kulcsos várakozási sorba
- A synchronized végén elengedi a kulcsot
- A felébredt alvónak (is) lehetősége van megszerezni a kulcsot és továbbmenni
- A synchronized blokkja végén ő is elengedi a kulcsot…

C#

A párhuzamos végrehajtás során az alábbi lépéseket kell megtenni

- Definiálunk egy függvényt, aminek nincsenek paraméterei és a visszatérési típusa void. Ez több rendszerben kötelezően run névre hallgat.
- Ennek a függvénynek a segítségével egy függvénytípust, delegáltat definiálunk. Könyvtári szolgáltatásként a ThreadStart ilyen, használhatjuk ezt is.
- Az így kapott delegáltat felhasználva készítünk egy Thread objektumot.
- Meghívjuk az így kapott objektum Start függvényét.

A párhuzamos végrehajtás támogatását biztosító könyvtári osztályok a System.Threading névtérben találhatók.

C#

```
using System;
using System.Threading;
class program {
    public static void szal() {
         Console.WriteLine("Ez a szálprogram!");
    }
    public static void Main() {
         Console.WriteLine("A főprogram itt indul!");
         ThreadStart szalmutato=new ThreadStart(program.szal);
           // létrehoztuk a  függvénymutatót, ami a szal() függvényre mutat
         Thread fonal=new Thread(szalmutato);
           // létrehoztuk a párhuzamos végrehajtást végző objektumot
           // paraméterül kapta azt a delegáltat (függvényt) amit  majd végre kell
            // hajtani
         fonal.Start();
           // elindítottuk a fonalat, valójában a szal() függvényt
           // a fonal végrehajtása akkor fejeződik be amikor  a szal függvényhívás
            // befejeződik
         Console.WriteLine("Program vége!");
         }
}
```

C#

Szálak vezérlése

- Sleep (millisec): statikus függvény, az aktuális szál végrehajtása várakozik a paraméterben megadott ideig.
- Join(): az aktuális szál befejezését várjuk meg
- Interrupt(): az aktuális szál megszakítása. A szál objektum interrupt hívása ThreadInterruptedException eseményt okoz a szál végrehajtásában.

C#

Szálak vezérlése

- Abort(): az aktuális szál befejezése, valójában a szálban egy AbortThreadException kivétel dobását okozza, és ezzel befejeződik a szál végrehajtása. Ha egy szálat abortáltunk, nem tudjuk a Start függvénnyel újraindítani, a start utasítás ThreadStateException kivételt dob. Ezt a kivételt akár mi is elkaphatjuk, és ekkor, ha úgy látjuk, hogy a szálat mégsem kell „abortálni”, akkor a Thread.ResetAbort() függvényhívással hatályon kívül helyezhetjük az Abort() hívását.
- IsAlive: tulajdonság, megmondja, hogy a szál élő-e
- Suspend(): a szál végrehajtásának felfüggesztése
- Resume(): a felfüggesztés befejezése, a szál továbbindul

C#

```
class program {
	public static void szal() {
		Console.WriteLine("Ez a szálprogram!");
		Thread.Sleep(1000);
		// egy másodpercet várunk
		Console.WriteLine("Letelt az 1 mp a szálban!");
		Thread.Sleep(5000);
		Console.WriteLine("Letelt az 5 mp szálban!");
		Console.WriteLine("Szál vég!");
	}
```

C# - .NET

```
class adatok
{
	public void mentes(string s)
	{
		Console.WriteLine("Adatmentés elindul!");
		for(int i=0;i<50;i++)
		{
		    Thread.Sleep(1);
		    Console.Write(s);
		}
		Console.WriteLine("");
		Console.WriteLine("Adatmentés befejeződött!");
	}
}
```

C# - .NET

```
class program {
        public static adatok a=new adatok();
        // adatmentésmező a programban
        // szál1 definiálás
        public static void szal1() {
                Console.WriteLine("Ez a szál1 program indulás!");
                // egy másodpercet várunk
                Console.WriteLine("Adatmentés meghívása szál1-ből!");
                a.mentes("+");
                Console.WriteLine("Szál1 vége!");
        }
        // szál2 definiálás
        public static void szal2() {
                Console.WriteLine("Ez a szál2 programindulás!");
                // egy másodpercet várunk
                Console.WriteLine("Adatmentés meghívása szal2-ből!");
                a.mentes("-");
                Console.WriteLine("Szál2 vége!");
        }
```

C# - .NET

```
public static void Main() {
        Console.WriteLine("A főprogram itt indul!");
        ThreadStart szalmutato1 = new ThreadStart(program.szal1);
        ThreadStart szalmutato2 = new ThreadStart(program.szal2);
        // létrehoztuk a függvénymutatókat,
        Thread fonal1=new Thread(szalmutato1);
        Thread fonal2=new Thread(szalmutato2);
        fonal1.Start();
        fonal2.Start();
        Console.WriteLine("Program vége!");
        }
}
```

Felváltva ír ki +-t és - -t a képernyőre!!

C# - .NET

Erőforráskezelés
A .NET keretrendszer számos osztályt és adattípust kínál számunkra, hogy a közös erőforrásokat kezelhessük. A CLR (Common Language Runtime) háromféle elérési módot biztosít globális változók, metódusok, osztályszintű metódusok, objektumok és blokkok számára:

- Szinkronizált kódterület - Szinkronizálható bármely osztályszintű és példányszintű metódus, vagy annak egy része Monitor használatával. Megjegyzendő, hogy nincs lehetőség statikus adattagok szinkronizációjára.
- Klasszikus kézi szinkronizáció - Számos szinkronizációs osztály használható arra, hogy összhangot teremtsünk a saját elvárásainknak megfelelően.
- Szinkronizált környezet A SynchronizationAttribute használata egyszerű, automatikus szinkronizációt valósít meg ContextBoundObject objektumokon. Minden objektum közös környezetben osztozik a záron.

C# - .NET

A Monitor osztály használatával elérhetjük, hogy egy kódrészletet egy adott időpontban csak egy szál használhasson.

A Monitor osztály minden metódusa statikus, így használatához nem kell példányosítani az osztályt.

A monitor indítása a Monitor.Enter(object) vagy a Monitor.TryEnter(object) metódushívással történhet.

Feloldása a Monitor.Exit(object) vagy a Monitor.Wait(object) hívással történhet.
Ha a monitor feloldása megtörtént, akkor a Monitor.Pulse vagy a Monitor.PulseAll hívás üzenetet küld a hozzáférési sorban álló szálaknak, hogy szabad az elérés.

C# - .NET

Amikor egy szál a lefoglalt kódrészletében Wait -et hív, akkor megszakad a hozzáférése, és bekerül egy várakozó sorba, majd a sor első elemét reprezentáló szál meghívja a Pulse vagy PulseAll metódust és az említett szál lép egyet a hozzáférési sorban.

Az Enter metódus atomi, így ha két szál egyszerre hívja ugyanarra a kódrészletre, akkor is csak az egyik kapja meg a jogot a futtatásra.

Javasolt a monitort belső objektumokon használni, mivel külső objektum zára deadlock -hoz vezethet. Javasolt, hogy a kódot egy try blokkban helyezzük el, és az Exit hívást pedig a finally ágban.

Ehelyett alkalmazható a lock(object) hívás, ami funkciójában megegyezik a Monitor -éval, de amikor a végrehajtás kilép a lock blokkjából, a zár feloldódik.

C# - .NET

Szinkronizáció:

1. A Synchronization() attribútummal kell jelölni az osztályt.
2. Az osztályt a ContextBoundObject osztályból kell származtatni.

Példa:

```
[Syncronization()]
class szinkronosztály: ContextBoundObject {
    int i=5; // egy időben csak egy szál fér hozzá
    public void Novel() // ezt a függvényt is csak egy
szál
    // tudja egyszerre végrehajtani
    {
          i++;
    }
…
}
```

C# - .NET

```
public void mentes(string s) {
	Monitor.Enter(this);
	Console.WriteLine("Adatmentés elindul!");
	for(int i=0;i<50;i++) {
		Thread.Sleep(1);
		Console.Write(s);
	}
	Console.WriteLine("");
	Console.WriteLine("Adatmentés befejeződött!");
	Monitor.Exit(this);
}
```

Az a blokk, amit a Monitor véd, megszakítás nélkül fut le – így az előző példában már nem lesz gond.

C# - .NET

És még sok-sok osztály segít – pl.

- WaitHandle
- Mutex, AutoResetEvent, ManualResetEvent
- InterLocked
- ThreadPool
- Timer
- Backgroundworker
- …

Go

Google nyelve

2007 – ben kezdték el fejleszteni

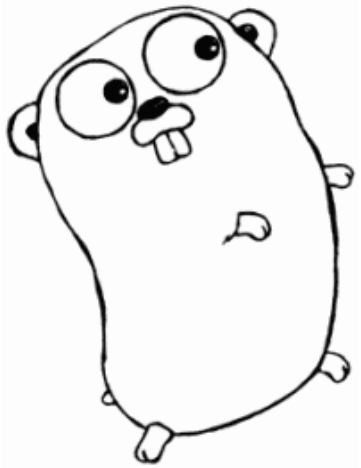
Első hivatalos verzió 2012. május 28-án jelent meg

Jelenlegi verzió: 1.10, 2018. május

Fordított, konkurens, imperatív programozási nyelv

C-szerű, statikus típusú nyelv

Szemétgyűjtés



Go – gorutinok

Osztott memória helyett csatornákon való kommunikáció

goroutine:

- Konkurens folyamat
- Közös címterületen, saját veremmel, ami dinamikusan nő
- Saját szemétgyűjtő
- A tényleges szálak kezelését elrejti a programozó elől
- A go kulcsszóval hívott függvény új goroutine-ban indul

Go – példa

```
import (
        "fmt"
        "time"
)
func IsReady(what string, minutes int64) {
    time.Sleep(time.Duration(minutes));
    fmt.Println(what, "is ready")
}
func main() {
    go IsReady("tea", 4);
    go IsReady("coffee", 2);
       // nem var  a befejezesre, ha o
       // befejezte, akkor kesz, ezert kell ide is egy Sleep...
    time.Sleep(time.Duration(6));
    fmt.Println("I'm ready...");
}
```

Go

A Go a párhuzamosságot osztott memória helyett adatcsatornákon történő üzenetküldésekkel támogatja.

- `chan // kétirányú csatorna`
- `<-chan // fogadó csatorna`
- `chan <- // küldő csatorna`

Az elemek típusa tetszőleges

Az inicializálatlan csatorna értéke nil

Új csatornát a make() függvénnyel hozhatunk létre

- `make(típus, kapacitás)`
- `make(chan int, 100)`

Csatorna típusnak kell lennie

Ha a buffer méret 0, akkor blokkol, amíg a fogadó goroutine nem áll készen fogadni, egyébként aszinkron

Go példák

Input-output csatorna:

- buffered_chan.go
- negyzetes.go
- sum_chan.go
- prime.go
- closurepl.go
- fibonacci_closure.go
- fibonacci_select.go

PVM - Parallel Virtual Machine

Szoftver rendszer, mellyel hálózatba kapcsolt számítógépeket egyetlen nagy párhuzamos számítógépként lehet látni és kezelni.

Kezdet: 1989 - Oak Ridge National Laboratory

A párhuzamos programok írásának egyik szabványává vált.

A publikus változata ingyen elérhető.

Sok hardver gyártó biztosítja a saját gépére optimalizált, gyorsabb változatát is.

Használata

PVM rendszerbe kapcsolhatunk több - akár különböző típusú - számítógépet.

A rendszer a rajta futó programok szempontjából ezek után egy nagy, elosztott memóriájú virtuális számítógépnek látszik.

Különböző processzorokon, különböző programokat indíthatunk el.

A szinkronizációt és a kommunikációt a PVM könyvtári függvényeivel oldhatjuk meg.

Működés

Számítógépek hálózatba szervezve

A felhasználó jogosult bármelyik gépre bejelentkezni

Minden gépen futtat egy pvm „démon”-t

És futtatja a taszkokat, melyek a démonokon keresztül lépnek egymással kapcsolatba

- Egy gépen több taszk is futhat

Elosztott rendszerek

Egy nagy virtuális címtér

- PVM
- Partitioned Global Address Space (PGAS) nyelvek és rendszerek

Üzenetküldés, elosztott memória

- Communicating Sequential Processes (CSP)
- Egymás memóriáját nem érik el, csak üzenetekkel kommunikálhatnak
- Message Passing Interface standard – legelterjedtebb
- Több millió folyamat (process) kezelésére alkalmas

C++11

std::thread szabványos könyvtár

thread osztály

Szálkezelő függvények

Kölcsönös kizárást biztosító osztályok: mutex és variációi: timed_mutex, recursive_mutex, recursive_timed_mutex, shared_timed_mutex

Érdemes megnézni:

http://en.cppreference.com/w/

C++11 thread példa

```
void egyik()
{
    cout << "Ez az egyik.\n";
}
void masik(int x)
{
    cout << "Ez a masik.";
    cout << " A parametere: " << x << "\n";
}
```

C++11 thread példa

A használata:

```
thread elso(egyik);
  // uj szalat indit, ami az egyik-et hivja
thread masodik(masik, 0);
  // uj szalat indit, ami a masik(0)-t hivja
cout << "main, egyik es masik most parhuzamosan
futnak ...\n";

// szalak szinkronizalasa:
elso.join();        // var, amig elso befejezodik
masodik.join();     // var, amig masodik befejezodik

cout << "elso es masodik lefutott.\n";
```

C++11 thread példa

```
mutex mtx;    // mutex a kritikus szakaszra

void masik_mutex(int x)
{
   mtx.lock();
   cout << "Ez a masik.";
   cout << " A parametere: " << x << "\n";
   mtx.unlock();
}
```

Az előző indítás helyett:

- `thread masodik(masik_mutex, 0);`

C++11 mutex példa

```
int main () {
  std::thread th1 (print_block, 50,'*');
  std::thread th2 (print_block, 50,'$');
  th1.join();
  th2.join();
  return 0;
}
```

C++11 mutex példa

```
#include <iostream> // std::cout
#include <thread>   // std::thread
#include <mutex>    // std::mutex
std::mutex mtx;     // mutex a kritikus szakaszhoz
void print_block (int n, char c) {
// kritikus szakasz (kizárólagos hozzáférés a std::cout –
hoz):
mtx.lock();
for (int i=0; i<n; ++i) {
std::cout << c;
}
  std::cout << '\n';
mtx.unlock();
}
```

Programkönyvtárak

Újrafelhasználható programkönyvtárak

Tervezési igények

- Az alap-követelmények ugyanazok, mint más szoftver tervezésénél
  - helyesség
  - hatékonyság
  - megbízhatóság
  - kiterjeszthetőség
  - stb.,
- De a könyvtár felé az igények erőteljesebbek!

Újrafelhasználható programkönyvtárak

Az újrafelhasználható könyvtárakban található osztályokkal kapcsolatban az alábbi tulajdonságok várhatóak el:

- Könnyű és intuitív használat
  - könnyen megérthető legyen, néhány használat után jól megjegyezhető
- Azonnali, széleskörű felhasználhatóság szoftverrendszerek széles körében
  - szolgáltatásokat nyújt
- A kurrens technológiához mért lehető leghatékonyabb megvalósítás

Újrafelhasználható programkönyvtárak

- A tesztelés körültekintősége és az osztály megbízhatósága
- Magas szintű dokumentáció
  - pontos, jól szervezett, hogy a felhasználó könnyen eligazodjon
- Platform-függetlenség
- Sokszor nyelv-függetlenség
- Meg kell hagyni a későbbi fejlesztés lehetőségét

Újrafelhasználható programkönyvtárak

Ezen követelmények néhány következménye:

- konzisztencia - a könyvtár minden komponense egy átfogó, koherens tervezésnek kell megfeleljen, és számos szisztematikus, explicit és egységes konvenciót kell kövessen.
- jó minőségű, homogén komponensek kellenek
- az objektum-orientált megközelítés az adatabsztrakció támogatása miatt különösen is alkalmas a könyvtárak készítésére

Újrafelhasználható programkönyvtárak

A fenti kritériumok egy részének egyidejű teljesítése igen nehéz.

- könnyen lehetséges, hogy a legáltalánosabb algoritmus nem a leghatékonyabb sok speciális, de gyakran előforduló esetben.

Újrafelhasználható programkönyvtárak

A könnyű felhasználhatóságnak különböző aspektusai vannak.

- Az egyik ezek közül az osztályok elkülöníthetősége.
  - Amennyiben nagyon sok hasonló osztály található egy könyvtárban, akkor a felhasználó számára igen nehéz a pontosan megfelelő kiválasztása.
- Általában elmondható, hogy előnyösebb kevés számú, majdnem teljesen ortogonális osztály definiálása
  - minden esetben egy jól meghatározott szerep betöltésére, s az optimális megvalósítást követve.

Tulajdonságok

A jó könyvtárfejlesztő tulajdonságai

- van absztrakciós készsége, képes arra, hogy az egyedi jelenségekből általánosítson
- van érzéke ugyanakkor a részletekhez
  - egy könyvtárban ui. minden kis tulajdonság számít.
    Itt nincs olyan, hogy 'elég jó', teljes kell legyen a könyvtár
- rend-mániákus - képes az osztályozásra, hogy mindennek megtalálja a helyét.

Tulajdonságok

Van irodalmi érzéke

- A leírásainak, a specifikációs részeket magyarázó megjegyzés-sorainak legyen stílusa és eleganciája

Elsőrendű programozó és tervező kell legyen

- Tudja átlátni a könyvtár későbbi használójának igényeit

"ego less" legyen

- Nem szabad, hogy egyéni stílus érvényesüljön, a cél a konzisztens stílus használatának lehetősége
- A kreativitása a könyvtár által nyújtott szolgáltatásokban jöjjön elő!

Programkönyvtárak jellemzése

Egy programkönyvtár osztályok gyűjteménye. Egy osztályt az általa nyújtott szolgáltatások jellemeznek.

Ezen szolgáltatásokat a következő két módon lehet csoportosítani:

- I.
  - Számított (rutin)
    - Függvény
    - Parancs
  - Tárolt
    - Attribútum

Programkönyvtárak jellemzése

Egy programkönyvtár osztályok gyűjteménye. Egy osztályt az általa nyújtott szolgáltatások jellemeznek.

Ezen szolgáltatásokat a következő két módon lehet csoportosítani:

- II.
  - Értéket visszaadó
    - Számított – függvény
    - Tárolt – attribútum
  - Parancs

Programkönyvtárak jellemzése

Osztályok másfajta csoportosítási lehetősége a csomagok (Java) vagy clusterek (Eiffel) használata.

A könyvtár és a csomag fogalmak 1-sok, illetve sok-1 kapcsolatban állhatnak egymással.

- Rendszerint a csomagok egy az egyben képződnek le fizikai tárolóterületre, például könyvtárszerkezetre.

Osztályhierarchia

OO esetben egy programkönyvtár elkészítése egy osztályhierarchia kialakítását is jelenti.

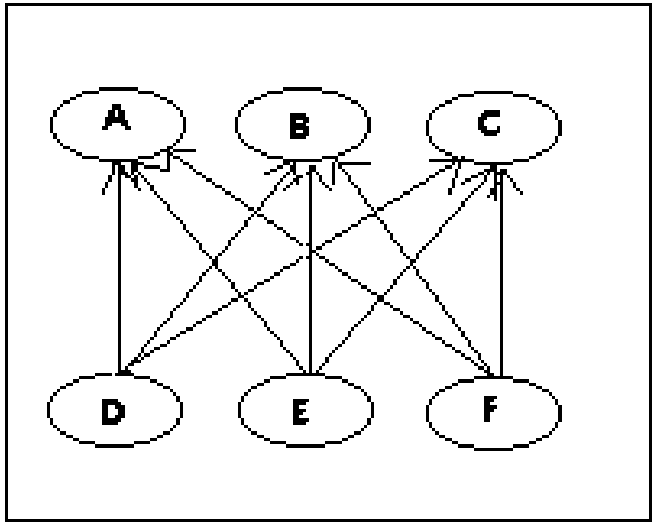
- A használhatóság szempontjából nem lényegtelen ezen hierarchia bonyolultsága vagy egyszerűsége.

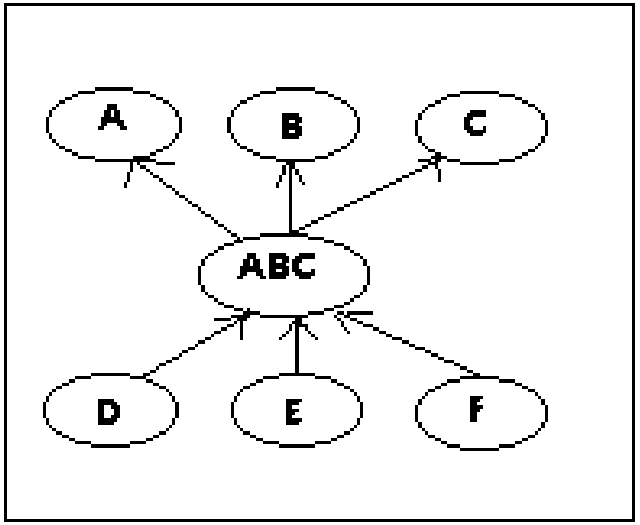
Az objektumorientált szemléletnek megfelelően igyekezzünk minél általánosabb osztályokat bevezetni (generalizáció) és az öröklődést felhasználva hozzuk létre a szükséges osztályokat (specializálódás).

- Ezáltal az osztályhierarchia karbantartása is leegyszerűsödik.

Osztályhierarchia

Új osztály bevezetésének másik oka az osztályhierarchia egyszerűsítése is lehet:





Biztonság

Egy sokak által használt programkönyvtár esetén fontos szempont a megbízhatóság és a biztonság.

Ennek biztosítása elő- és utófeltételek, valamint invariánsok adásával valósítható meg.

Ez – a programkönyvtár írói és használói között – egy szerződés létrejöttét jelenti, ahol az előfeltétel a felhasználó kötelezettsége és a programozó biztosítéka, míg az utófeltétel a programozó kötelezettsége és a használó biztosítéka.

Osztályhierarchia

Így a programozónak nem kell "defenzíven" programoznia, (minden lehetséges hibalehetőségre és bemenetre felkészülve) és a kód is hatékonyabb és egyszerűbb lesz.

A lehetséges kétfajta megközelítési mód

- Toleráns megközelítés: a programkönyvtári rutinoknak nincs (vagy csak gyenge) előfeltételük és minden lehetséges bemenetre valamilyen módon reagálnak.
- Követelődző megközelítés: minden rutin szigorú előfeltételekkel rendelkezik, ezek biztosítása a felhasználó felelőssége.

Az ajánlott megközelítés:

- Csak az absztrakt művelet logikai helyes elvégzéséhez szükséges előfeltételeket ellenőrizzük.
- Csak azt ellenőrizzük, ami, ha nem teljesülne, súlyosan befolyásolná a hatékonyságot.

Az így létrejött szerződés bármilyen megsértése programhibát jelez.

- Az előfeltétel megsérülése felhasználó oldali hibát, míg az utófeltétel vagy az invariáns sérülése programkönyvtár hibát jelez.
- Ezért a programkönyvtár fejlesztése után a terjesztés előtt a hatékonyság érdekében elég csak az előfeltétel teljesülés vizsgálatát bekapcsolni.

A lényeg

Explicit megszorítás kell, hogy a felhasználó tudja, mire számíthat!

Dokumentáció

Egy programkönyvtár több fejlesztő munkája.

A felhasználhatóság szempontjából ezért igen fontos a jó dokumentáció.

Lehetséges megközelítési módok:

- Forrásszöveg?
- Különálló dokumentáció?

Dokumentáció

A forrásszöveg nem elég absztrakt.

- A felhasználó számára fontos információkon kívül az alacsonyszintű implementációt is tartalmazza
- A használata túl sok információ átfésülését jelentené, vagy arra sarkallná a programozókat, hogy nem publikusnak szánt implementációs lehetőségeket is kihasználjanak.
- Előnye, hogy mindig aktuális.

Dokumentáció

Különálló dokumentáció esetén nem biztosított a szoftver és a dokumentáció konzisztenciája.

Előnye, hogy csak a felhasználó számára fontos dolgokat tartalmazza.

Dokumentáció

Belső dokumentáció elve : a szoftver dokumentációja a programszövegbe legyen beágyazva.

Ennek előnyei:

- Forrásszöveg és dokumentáció mindig konzisztens marad.
  - ?!
- Dokumentáció kinyerése automatizálható
  - javadoc

Eiffel flat-short forma

- az osztálynak egy olyan verziója, ami tartalmazza az öröklött és az itt bevezetett jellemzőket, figyelembe véve minden átnevezést és átdefiniálást, felépítve a teljes elő- és utófeltételt és a típusinvariánst.

Karbantartás

Mivel egy programkönyvtár állandó fejlesztés alatt áll, ezért lehetőséget kell biztosítani újabb verziók problémamentes beépítésére.

A felhasználás szempontjából a következő változtatások nem jelentenek problémát:

- Osztály bővítése új szolgáltatással.
- Egy szolgáltatás implementációjának lecserélése.
- Előfeltétel gyengítése, illetve utófeltétel szigorítása.

Karbantartás

A következő esetekben lehet szükség egy szolgáltatás új verziójának bevezetésére

- Szolgáltatás nevének megváltoztatása
  - Például egységes névkonvenció kialakítása miatt
- Jobb szignatúra vagy specifikáció megadása.
- Rutin elavulttá nyilvánítása.
  - Ez történhet nyelvi kulcsszavak használatával (Eiffel: obsolete, Java: deprecated), vagy a régi változat átírásával oly módon, hogy az egyből az új változatot hívja meg (hívásátirányítás), esetleges figyelmeztető üzenet kiírása kíséretében.

Osztályok mérete

Egy osztály méretét a következő módokon lehet meghatározni:

- Sima (teljes) méret = szolgáltatások száma (öröklött + itt bevezetett).
- Közvetlen méret = közvetlen (nem absztrakt, illetve újradefiniált) szolgáltatások száma.
- Növekményes méret = közvetlen és újradefiniált szolgáltatások száma.

Osztályok mérete

Másik szempont lehet, hogy tartalmazza-e a nem exportált jellemzőket:

- Külső méret
- Belső méret

Osztályok mérete

A teljes méretre nyilván nincs felső határ, a közvetlen, ill. a növekményes méret az, ami szerepel az osztály szövegében.

Egy adott méreten felül az osztály kezelhetetlen.

Két megközelítési lehetőség

- Minimalista nézőpont: egy programkönyvtár osztálya a megvalósított típusnak csak a legalapvetőbb (atomi) szolgáltatásait tartalmazza, redundáns (azaz ami atomi szolgáltatások használatával is megvalósítható) metódusokat pedig nem.
  - Ez persze a programkönyvtár írójának kedvez, hiszen kevesebb munkát és egyszerűbb karbantartást jelent.

Osztályok mérete

- Bevásárlólista nézőpont: minden olyan szolgáltatást vegyünk bele az osztályunkba, ami az alábbi szempontoknak megfelel:
  - A szolgáltatás beleillik az absztrakt adattípus megvalósítási sémájába.
  - Nem rontja el az osztály helyességét, azaz az osztályinvariánst.
  - Hasznos funkcionalitást valósít meg.
  - Nem duplikál valamely már meglévő szolgáltatást.
  - Megfelel a könnyen kezelhetőség kritériumainak (lásd később).

Osztályok mérete

A gyakorlatban általában

- a programozási nyelvtől elvárt szempont a minimalista nézőpont, azaz hogy műveletet csak minél kevesebb féleképpen lehessen hatékonyan elvégezni,
- míg a programkönyvtárak tervezésekor a felhasználás megkönnyítése érdekében inkább a bevásárlólista nézőpont érvényesül.

Szolgáltatások száma

Túl sok szolgáltatás a programkönyvtár használatának megtanulhatóságát is kedvezőtlenül befolyásolja, ennek kezelhetősége érdekében a következő rendezőelvek betartása ajánlott:

- Minden szolgáltatás egy jól meghatározott szerződéssel definiált, csak a specifikációjuk, nem pedig az implementációjuk alapján kerülnek felhasználásra.
- Az osztálydokumentáció egységes és könnyen áttekinthető formátumú legyen, feltüntetve minden szolgáltatás szerződését és elrejtve az implementációs részleteket.

Szolgáltatások száma

- A szolgáltatások nevei szigorú elnevezési konvenciót kövessenek.
- A szolgáltatásokat csoportosítsuk pontosan definiált kategóriák szerint, ezen kategóriák minden osztálynál egyezzenek meg, sorrendjük a dokumentációban legyen azonos, kategórián belül a szolgáltatások névsorrendben következzenek.

Példaadatok

Általában kijelenthető, hogy 80 szolgáltatásnál többet tartalmazó osztály esetén már meggondolandó, hogy nem lehetne-e az osztályhierarchiát tovább bontani.

Szolgáltatások mérete

Mivel a programkönyvtárak használata a megvalósított szolgáltatások meghívásából áll, ezért igen fontos, hogy a használó pontosan ismerje egy szolgáltatás paramétereinek számát, típusát és sorrendjét.

- Ezért a felhasználás szempontjából egy szolgáltatás méretét tulajdonképpen a megadandó paraméterek száma jellemzi.

Ez a méret erősen függ a programkönyvtár feladatától is, hiszen egy grafikus felületet megvalósító, vagy statisztikai számításokat végző osztály metódusai rengeteg argumentumot igényelhetnek.

Szolgáltatások mérete

Nagyszámú argumentum esetén érdemes azokat a következő két csoportra osztani:

- Paraméter: olyan argumentum, amelynek értékével valamilyen műveletet kell elvégezni.
- Opció: olyan argumentum, amely akár el is hagyható, ugyanis rendelhető hozzá egy alapértelmezett érték.
  - Csak a paraméterek feldolgozását befolyásolja.

Egy szolgáltatás fejlesztése során a paraméterlista lehetőleg ne változzon, míg új opciókat nyugodtan fel lehet venni.

Szolgáltatások mérete

Ezekből következik, hogy a szolgáltatás méretének csökkentése érdekében csak paramétereket vegyünk fel az argumentumlistába.

Az opciókat pedig külön attribútummal reprezentáljuk és adjunk hozzá új beállító/lekérdező szolgáltatásokat.

- Ez az ún. opció beállítási technika.
- Ez persze felveti az új opció-attribútum globalitásának kérdését, sőt növeli az osztály méretét is.

Példaadatok

Általában kijelenthető, hogy az a jó, ha a szolgáltatások átlagos paraméterszáma 0.5, és 5-nél több paraméter esetén meggondolandó, hogy nem lehetne-e a szolgáltatást tovább bontani.

A szolgáltatások lekérdezés – parancs javasolt aránya

- 60 - 40%

Elnevezés

Egy programkönyvtár használatát nagyban elősegíti az egységes elnevezési konvenció.

Ez konzisztens névhasználatot jelent, kialakítása padig azon alapul, hogy kategorizáljuk a megvalósított adattípusok közös tulajdonságait

Elnevezés

Például

- mechanizmus egy adott elem elérésére (get)
- mechanizmus egy elem megváltoztatására (put)
- mechanizmus egy elem törlésére (remove)

Így azonos neveket használunk különböző osztályokon belül, de ez mégsem vezet félreértéshez, mert a típusok különbözősége miatt általában a szignatúrák sem egyeznek meg.

Elnevezés

Ilyen elnevezés-konvenció bevezetésekor szinte biztos, hogy használni fogjuk a következő standard elnevezéseket:

- `capacity` - adott struktúra kapacitását adja vissza.
- `count` - adott struktúra elemszámát adja vissza.
- `empty` - megadja, hogy az adott adatstruktúra üres-e.
- `full` - megadja, hogy az adott adatstruktúra tele van-e.
- `get` - struktúra adott elemét adja vissza.
- `has` - lekérdezi, hogy adott elem a struktúra eleme-e.
- `put` - struktúra adott elemét állítja be.
- `remove` - struktúra adott elemét kitörli.

Elnevezés

Irányelvek:

- A név legyen rövid (rendszerint egy szó), de beszédes.
- Szolgáltatás neve ne tartalmazzon utalást a szolgáltatást tartalmazó osztályra
  - kivéve többszörös öröklődés esetén, ha átnevezés kell
- Osztály neve mindig főnév(i szerkezet) legyen.
- Parancsok (eljárások) nevei legyenek (felszólító módú) igék esetleges főnévi vagy értelmező jelzői kiegészítővel.
- Nem logikai lekérdezések nevei legyenek főnevek vagy főnévi szerkezetek.

Elnevezés

- Logikai lekérdezés neve legyen olyan melléknév, amely igen/nem választ sugall, vagy használjuk az is_ (-e) szerkezetet.
- A szolgáltatás nevét a célobjektum szemszögéből lehessen értelmezni, mivel egy szolgáltatás hívásakor mindig létezik egy objektum (a célobjektum) aminek a szolgáltatásáról szó van.
  - has
- összetartozó művelet és lekérdezés párok elnevezéseinek szótöve legyen azonos
  - extendible - extend

Az osztályok fajtái

- Konkrét típusok
- Absztrakt típusok
- Csomópont típusok
- Felület osztályok
- Kövér interfészek
- Alkalmazás keret
- Leíró osztályok

Az osztályok fajtái

Konkrét típusok

- Minden programkönyvtár hierarchiában találunk olyan típusokat, melyek alapvető adatstruktúrákat reprezentálnak (lista, vektor, string, dátum, komplex szám…).

Az osztályok fajtái

A következő jellemzi:

- Szoros illeszkedés egy konkrét fogalomhoz és a megvalósítás módjához.
- Érthetőség, önálló használhatóság.
- Erősen implementációfüggők, ezért hatékony a megvalósításuk, de bármilyen módosításukkor minden felhasználói kódot újra kell fordítani.
- Minimális mértékben függenek más osztályoktól.
- Különállóan (izoláltan) is fordíthatóak és használhatóak, ugyanakkor nem, illetve ritkán lehet tőlük örökölni.

Az osztályok fajtái

Absztrakt típusok

- Szerepük rendszerint egy adott specifikáció (akár interfész is lehetne) bevezetése. Pl. halmaz
- Az absztrakt típusokat konkrét típusok segítségével lehet implementálni.

Tulajdonságaik:

- Azonos elérési módhoz többfajta implementáció is létezhet.
- Hatékony tárkihasználás és elfogadható futási idő a virtuális metódusok segítségével.
- Az implementáció minimális mértékben függ más osztályoktól.
- Különállóan is lefordíthatóak – érthetőek önmagukban.

Csomópont típusok

Az öröklési hierarchia belső pontjait alkotják.
Jellemzőik:

- Az ősosztályok szolgáltatásait implementálja, ugyanakkor annak interfészét kibővíti virtuális metódusokkal is, melyeket maga is implementál.
- Függ az őseitől.
- Lehet belőle örökölni.
- Példányosítható.

Példa: ablak, négyszög...

Felületosztályok

Egy felület igazítása úgy, hogy jobban illeszkedjen a felhasználó elvárásaihoz.

- Pl. vektor ne 0-tól legyen indexelve.

Ellenőrzött vagy korlátozott felületek biztosítása.

„Kövér interfészek”

Ez egy olyan típus, amely nem függ más osztályoktól, rengeteg szolgáltatást vezet be

- ezek közül csak a legalapvetőbbekhez ad implementációt, míg a speciálisabb szolgáltatásokat virtuálisként deklarálja
- hogy lehessen példányosítani, ad hozzá egy üres implementációt, ami rendszerint egy hibaüzenet kiírását jelenti
  - jelzi, hogy az adott szolgáltatás nem implementált.
  - Példa: általános tároló (container) osztály.

Alkalmazás keretek (frame)

Ezen absztrakt osztályok tulajdonképp egy mini alkalmazást valósítanak meg.

- Maga a programlogika van csak az osztályon belül implementálva, minden egyéb paraméter-bekérő és alapműveletet elvégző metódus virtuális.
- Példa: szűrő osztály, amely egy bemeneti adatfolyam minden elemén végiglépkedve adott feltétel teljesülésekor bizonyos műveleteket elvégez.
- A bemeneti adatfolyamon történő végigléptetés, adott tulajdonság fennállásának ellenőriztetése, illetve a kívánt művelet meghívása és esetleges hibakezelés ezen osztályban van implementálva.

Leíró osztályok

Míg egy absztrakt interfész és annak konkrét implementációja közötti kapcsolat a program futása során állandó, az implementáló osztály mérete is állandó, felmerülhet az igény adott interfészen keresztül változó méretű osztályok objektumainak kezelésére is.

- Erre ad megoldást a leíró osztályok használata.
- Ekkor ugyanis az implementáció két részre bomlik: az aktuális reprezentációra és a reprezentáció elérését biztosító leíró (handle) objektumra.
  - Ez tulajdonképp a mutató típus objektumorientált megvalósításának tekinthető.
- Példa: fájlleíró osztályok

Memóriakezelés

A programkönyvtár tervezés egyik kulcsfontosságú kérdése.

Két alapvető probléma:

- Memóriafoglalás új objektumok létrehozásakor.
- Memória felszabadítása.

Míg az első az operációs rendszer feladata, addig a második problémára két megoldási irányzat is született:

- Automatikus szemétgyűjtés (pl. Java). Ez rendszerint referenciák bevezetését és a mutató típus megszüntetését jelenti.
- Objektumok kézzel történő felszabadítása. Hatékonyabb, de sokkal veszélyesebb.

Fontos tanácsok

How To Write Unmaintainable Code

- https://thc.org/root/phun/unmaintain.html